Frank DENIS random thoughts.

Performance of WebAssembly runtimes in 2023

Using libsodium in a web browser has been possible since 2013, thanks to the excellent Emscripten project.

Since then, WebAssembly was introduced. A more efficient way to run code not originally written in JavaScript in a web browser.

And libsodium added first-class support for WebAssembly in 2017. On web browsers supporting it, and in allowed contexts allowing it, that gave a nice speed boost. Like JavaScript, the same code could seamlessly run on multiple platforms.

Also like JavaScript, applications started to use WebAssembly server-side. Still like JavaScript, and ignoring bugs in runtime implementations, it doesn't allow untrusted code to read or write memory outside of a sandbox. That alone makes it a compelling choice for application plug-ins, function-as-a-service services, smart contracts and more.

In 2019, support for a new WebAssembly target (`wasm32-wasi`) was added to libsodium, making it possible to use the library outside web browsers, even without a JavaScript engine.

As of today, multiple runtimes support `wasm32-wasi`, but on the same platform, the same code can run with very different performance across runtimes.

Benchmarking abilities for `wasm32-wasi` were thus added to libsodium.

This benchmark proved to be more representative of real-world performance than micro-benchmarks. Sure, libsodium is a crypto library. But the diversity of the primitives being measured exercises the vast majority of optimizations implemented (or not) by WebAssembly runtimes/compilers/JITs, and this benchmark turns out to be a good representative of real-world applications.

Since its introduction, the libsodium benchmark has been widely used by runtimes to improve their optimization pipelines, by researchers to measure the impact of experiments on WebAssembly, and by users to pick the best runtimes for their workload.

But it's been a while since results were published here. Meanwhile, runtimes have improved, so an update was overdue.

What happened since the last benchmark

- InNative is still actively maintained, but still didn't get `WASI` support. While it looks worth benchmarking, this is unfortunately still a showstopper for the time being.
- Lucet has been EOL'd, as Wasmtime and Wasmer provide the same features, using the same code generator.
- Fizzy doesn't seem to be actively maintained any more.
- WAVM doesn't seem to be maintained any more either. WAVM implemented proposals before everybody else, and has consistently been the fastest runtime. Has EPIC Games given up on WebAssembly? Minor updates keep being made in the fork used by FAASM, but there's no more activity in the upstream repository. This is a big loss for the WebAssembly community.
- SSVM became WasmEdge, the runtime from the Cloud Native Computing Foundation. The project is very active, and has been focused on performance since day one. It comes with a lot of features, including the ability to run plug-ins.
- Wasm3's development pace seems to have slowed down. However, it remains the only WebAssembly runtime that can easily be embedded into any project, with minimal footprint, and amazing performance for an interpreter. It still doesn't have any competition in that category.
- Wasmtime quickly went from version 0.40 to version 3.0.1, with version 4 being round the corner. Every release is an opportunity to update Cranelift, the code generator it is based on. A lot of improvements were recently made to Cranelift, so it's high time to see how they reflect in benchmarks.
- Wasmer kept releasing unique tools and features, such as the ability to generate standalone binaries. Their single-pass compiler also got updated. Let's put it to the test!
- Wamr saw a bunch of new releases. Pre-built binaries are also now available. This is very good news, as the compilation process used to be tedious.
- Wazero is a new, zero-dependency runtime for Go. While it hasn't reached version 1.0 yet, it looks extremely promising.
- Node has excellent support for WebAssembly and `WASI`. Yet, the latter still isn't enabled by default, and requires command-line flags (`--experimental-wasm-bigint --experimental-wasi-unstable-preview1`).
- Bun showed up as a modern alternative to `node`, based on JavaScriptCore. It doesn't support `WASI` out of the box, but wasmer-js emulates it perfectly.
- GraalVM now includes experimental support for WebAssembly.

Most other runtime projects (Asmble, Wac, Windtrap, Wagon, Py-Wasm, …) appear to have been abandoned.

Compiling C code to WebAssembly

WebAssembly includes several proposals to improve performance, such as the ability to perform tail-call optimizations, support for 128-bit vectors, threads, and bulk memory operations (`memcpy`/`memset`).

Unfortunately, all these interesting additions are still only proposals that runtimes may or may not implement, and may or may not enable by default. An application compiled with support for one of these proposals will crash on a runtime that doesn't support it, or didn't enable it.

Ideally, every WebAssembly application should be distributed for multiple targets (`wasm32-wasi-generic`, `wasm32-wasi-generic+simd128`, `wasm-wasi-generic+simd128+threads+tail_calls`, etc). But this is unmanageable, and runtimes don't even implement the ability to automatically select a suitable target.

So, in practice, to write applications that are compatible with a wide range of runtimes, using these proposals is not an option. Bulk memory operations may be an exception: they are supported by the vast majority of runtimes, and enabled by default in `wasm3`, `wasmedge`, `wasmer`, `wasm2c` and `node`.

To benchmark the runtimes presented above, we need to build the library for the `wasm32-wasi` flavor of WebAssembly.

In order to do so, and since libsodium is written in C, we have to use a C/C++ (cross-)compiler that can target `wasm32-wasi`: the `zig cc` command from the Zig toolchain.

Previously, compiling libsodium to `wasm32-wasi` required setting the compiler to the `zig cc --target=wasm32-wasi` command. But the library source code now includes a Zig build file that can be used as an alternative to `autotools`. So, for this benchmark, the library was built with the following command:

```
zig build \
  -Dtarget=wasm32-wasi \
  -Drelease-fast \
  -Denable_benchmarks=true \
  -Dcpu=generic+bulk_memory
```

The Zig version used was `0.11.0-dev.863+4809e0ea7`, and libsodium was at revision `58ae64d319246e5530c`.

Along with the library itself, the resulting `zig-out/bin/` folder contains WebAssembly files for every benchmark.

Methodology

Tests output the time it took for them to run. So, the benchmark ignores the setup/teardown time individual runtimes may have. Similarly, the compilation time required by ahead-of-time compilers is intentionally not measured. We only measure actual execution performance.

The benchmark was run on a Zen 2 CPU provided by Scaleway. Nothing else was running on the instance, tests were pinned to a single CPU core, and each test was run 200 times in order to further reduce noise.

Prior to being run, the WebAssembly files were further optimized with the `wasm-opt -O4 --enable-bulk-memory` command (part of the `binaryen` tools).

The exact set of WebAssembly files used for the benchmark can be found here.

The following runtimes have been benchmarked:

- `iwasm`, which is part of the `WAMR` ("WebAssembly micro runtime") package - pre-compiled files downloaded from their repository
- `wasm2c`, included in the Zig source code for bootstrapping the compiler
- `wasmer` 3.0, installed using the command shown on their website. The 3 backends have been individually tested
- `wasmtime` 4.0, compiled from source
- `node` 18.7.0 installed via the Ubuntu package
- `bun` 0.3.0, installed via the command show on their website
- `wazero` from git rev `796fca4689be6`, compiled from source

Unfortunately, GraalVM couldn't take part of the benchmark, as its support for `WASI` appears to be very limited. The absence of `random_get()` required to generate random numbers was a blocker. It also doesn't support bulk memory operations. Maybe next time?

Tests have also been grouped by category. This significantly improves readability compared to previous iterations of the benchmark.

Results have been median normalized. The X-axis represents how much slower a runtime is compared to the median performance (`1`). So, smaller is better, and a result of `2` means that the runtime was 2x slower than the median.

AEAD benchmark

The authenticated encryption tests feature a small function called many times (for every input block) with different parameters. Performance typically depends on auto-vectorization and register allocation.

None of the contestants appears to have been able to auto-vectorize anything, so results are fairly similar.

Authentication benchmark

These tests are based on a software implementation of the SHA-2 (SHA-256, SHA-512, SHA-512/256) hash function.

Like AEADs, they rely on a function called many times with different parameters. However, there are also quite a lot of constants, that compilers have an opportunity to inline.

In these tests, `wasmedge` doesn't perform well compared to Wasmer with the LLVM backend, which is unexpected considering the fact that they are both based on LLVM.

The `JavaScriptCore` engine, represented by `bun`, shows some serious room for improvement on these tests. It seems to lack optimization passes that even single-pass compilers have.

Box benchmark

These tests combine bitwise operations and arithmetic. Results are fairly similar, with the exception of single-pass compilers that cannot afford expensive optimizations.

Arithmetic over elliptic curves

A lot of arithmetic, mostly using 64-bit registers. Unsurprisingly, results are similar to the box tests.

Hashing benchmark

BLAKE2B and SHA-2 hashing. This is similar to the authentication benchmark.

`bun` remains slower than other runtimes, even though BLAKE2B doesn't use large constant tables like SHA-2 does. There's definitely a pattern found in hash functions that `JavaScriptCore` cannot properly optimize yet.

Key exchange and key derivation benchmark

Same hash functions under the hood, but with a major difference in the input of these functions: here, inputs are short. Ignoring single-pass compilers, results are fairly close.

Metamorphic benchmark

Apparently not a lot of wiggle room for optimizations here: with the usual exceptions, all runtimes perform almost exactly the same, with the exception of Wazero.

These tests perform memory allocations and require a lot of random numbers (obtained via a `WASI` syscall). This is what may cause Wazero to be so slow, rather than how the code is optimized.

One-time authentication benchmark

This is benchmarking the Poly1305 function. This is simple arithmetic using 64-bit registers. Multi-pass compilers seem to apply the same optimizations; so do single-pass compilers.

Password hashing benchmark

Password hashing functions rely on memory reads and writes in unpredictable locations. There's probably not much to optimize beyond what the C compiler already did.

Diffie-Hellman benchmark

Arithmetic over finite fields. Optimization opportunities include usage of `ADCX/ADOX` for carry propagation.

Overall, all runtimes perform quite well, with the exception of Wazero, which is significantly slower than Wasmer with the singlepass backend.

Secretbox benchmark

Here, we benchmark authenticated encryption using salsa20, chacha20 and poly1305. Simple arithmetic on things that mostly fit on 32-bit and 64-bit registers.

Keygen benchmark

These tests are all about randomness extraction. There's not much to optimize here. The tests mainly measure the overhead of `WASI` calls to get random bytes, and, very likely, the over of `WASI` calls in general.

The difference between `node` and `bun` is expected, as `node` has native support for `WASI`, while `bun` requires the `wasmer.js` emulation layer. It shouldn't take long for `bun` to also include native `WASI` support, which should bring it in the same ballpark as other runtimes.

`wasmedge`, however, doesn't have the `WASI` emulation excuse. Calling external functions in `wasmedge` may have more overhead than with other runtimes.

Signature benchmark

Arithmetic over elliptic curves, triggering common optimization passes, and making very few `WASI` syscalls. Here, `wasmedge`, also based on `LLVM` manages to be faster than `wasmer` with the `LLVM` backend.

Utilities benchmark

Codecs, comparisons functions, and other simple helper functions, designed to intentionally prevent compiler optimizations, as a best effort to make them constant-time.

As expected, performance is very similar across runtimes.

Stream ciphers benchmark

Salsa20 and Chacha20 stream ciphers, so, mostly bitwise arithmetic involving vectors of 32-bit values. There are opportunities for auto-vectorization, but this is not trivial.

Performance is very similar, with the exception of single-pass compilers.

Cumulative benchmark

`iwasm`, `wasmer` (LLVM backend) and `wasmedge` are all using `LLVM` for code generation. Intuitively, one may expect very similar performance. But this is not the case.

But `iwasm`, a newcomer in this benchmark (at least in AOT mode), was consistently the fastest runtime. This is a pleasant surprise. Its performance is likely to be similar to `wavm`, that it can be considered as replacement for.

That being said, `wasmer` is very close.

In the `LLVM` category, I was expecting `wasmedge` to take the lead given how much performance-focused the project has been since the beginning. Its results were a little bit disappointing. `V8` (represented by `node`) is about as fast.

On the other hand, runtimes using `cranelift` for code generation (`wasmer` with that backend, and `wasmtime`) perform virtually the same.

Zooming out a little bit, what is really impressive is how close the `LLVM`-based results and the `cranelift`-based results are.

That's right. The `cranelift` code generator has become as fast as `LLVM`. This is extremely impressive considering the fact that `cranelift` is a relatively young project, written from scratch by a very small (but obviously very talented) team.

The `JavaScriptCore` engine, represented by `bun`, was disappointing. As a JavaScript engine, it is blazing fast. As a WebAssembly engine, not so much yet. Some of these results can be explained by the lack of a native `WASI` implementation, but there's probably more room for optimization.

`wasmer` with the `singlepass` backend and `wazero` have in common that they compile very quickly, in a streaming fashion. This is a critical property for some applications involving untrusted inputs.

The flipside is that expensive optimizations cannot be done. This is an inevitable tradeoff.

`wazero`'s results are not great yet. But with its seamless integration with the Go language involves additional constraints, especially on Intel CPUs that have a limited number of registers. `wazero` may perform much better on ARM CPUs, and this is something that we will measure soon.

In any case, the `wazero` team is taking performance very seriously, and already started investigating these results.

Comparison against native code

How does WebAssembly compare against native code?

The chart above represents the ratio between the time taken to run individual tests with the fastest WebAssembly runtime, and the same tests with the library compiled as native code, with architecture-specific optimizations.

Two outliers have been hidden: the `aegis128l` and `aegis256` tests. Native code takes advantage of native CPU instructions to compute AES rounds. WebAssembly, on the other hand, cannot take advantage of these instructions and has to fall back to slow, software implementations.

As a result, these AES-based tests were 80 times slower than native code when running WebAssembly. This is not representative of most applications. However, it highlights a real limitation of WebAssembly for cryptographic operations relying on AES.

Ignoring this, when using the fastest runtime, WebAssembly was only about 2.32 times slower (median) than native code with architecture-specific optimizations. Not bad!

Verdict

We have four classes of WebAssembly runtimes:

- Interpreters, and in that category, `wasm3` is probably still the best option. It's also far easier to embed than anything else.
- `LLVM`/`Cranelift`/`V8`-based runtimes with comparable performance.
- `JavaScriptCore`-based runtimes – slightly behind the rest *yet*.
- Single-pass compilers.

iwasm

If you're looking for the best performer, `iwasm` is currently the one to choose.

`iwasm` is part of the `WAMR` project.

Compared to other options, it is intimidating.

It feels like a kitchen sink, including disparate components (IDE integration, an "application framework library", remote management, an SDK) that makes it appear as a complicated solution to simple problems. The documentation is also a little bit messy and overwhelming.

It also has a lot of knobs, both at compile-time and at run-time. The memory usage tuning page is scary. I wish there was just a `max_memory` knob, rather than multiple settings that may or may not work depending on applications and how they were compiled. That being said, this also applies to other runtimes, and the default values are probably reasonable.

Ignoring this, there is a simple C API, as well as bindings for Python and Go. So in these languages, it is actually quite easy to use.

The runtime itself is very small (50 KB) and has a small memory footprint. It can be tailored to applications in order to reduce it even further, making it an excellent choice for constrained environments. Especially since it supports platforms such as Zephyr, VxWorks, NuttX and RT-Thread out of the box.

LLVM/Cranelift/V8-based runtimes

`node`, `wasmtime`, `wasmedge` and `wasmer` are in the same ballpark.

None of them has a stable API yet. This is especially true for Rust APIs. If your application depends on the Rust APIs of `wasmer` or `wasmtime`, be prepared for maintenance costs.

However, the C APIs are far more stable: `wasmtime`'s C API only had one recent breaking change, while `wasmer` completely revamped the API once, and kept it stable since.

The `wasmedge` C API is very different, and has many more features. Breaking changes still happen, but they are generally minor and easy to deal with.

`node` (or, actually, `V8`) has by far the most stable API, even though `WASI` support in `node` is still considered experimental.

So, `node` (or `V8`+`uvwasi` when embedded) is a fine, safe, conservative choice, and for WebAssembly, there's not a lot of performance to gain in switching to something else. And yes, like others, `V8` supports full module AOT compilation, too.

However, in order to run WebAssembly code, the entire `V8` engine has to be shipped. This is a huge dependency. Not an issue for applications also needing support for JavaScript, but this is definitely overkill for applications that only need WebAssembly.

This is where `wasmtime`, `wasmedge` and `wasmer` come into play. They are smaller (= reduced attack surface), and include features suitable for running third party code/smart contracts, such as gas metering.

For most users, there are no significant differences between these three runtimes. They share similar features (such as AOT compilation) and run code the same way, roughly at the same speed.

They have some unique features that can make a difference to some applications, though.

`wasmer` has the largest ecosystem, with libraries making it very easy to use from many programming languages, as well as applications such as PostgreSQL and Visual Studio Code.

It also includes the ability to generate standalone binaries for all supported platforms, has a package manager and more. In spite of some breaking changes, it seems more focused on API stability than alternatives, making it a reasonable choice for applications that are planned to be maintained for a long time.

`wasmedge` is the runtime from the CNCF, and what Docker uses to run containers with WebAssembly applications. So, even if this is not your runtime of choice, testing that your WebAssembly code properly runs on it is highly recommended, as its popularity is likely to skyrocket.

`wasmedge` comes with networking support, although this is currently limited to Rust and JavaScript applications. But one of the best features of `wasmedge` is its plug-in system, allowing it to be extended without having to change the core runtime. Out of the box, it comes with plug-ins implementing the `wasi-nn` and `wasi-crypto` proposals, as well as an HTTP client plug-in and a simple way to run external commands in a controlled way.

`wasmedge` includes libraries to easily embed it into C, Rust, Go, JavaScript and Python applications. It supports preemption (this is nice!), instruction counting and gas measuring/limiting,

CVEs for quite a few vulnerabilities in wasmtime have been assigned. Some are severe (use after free, out-of-bound accesses, type confusion…) possibly leading to secret disclosure and arbitrary code execution.

Is it a bad thing? Actually not. To my knowledge, `V8`, `JavaScriptCore` and `wasmtime` are the only WebAssembly runtimes having publicly disclosed vulnerabilities so far.

Which is quite surprising, especially since some of these vulnerabilities were in `cranelift`, which is also used by `wasmer`.

`wasmtime` did responsible security disclosure after every vulnerability, and went above and beyond to prevent similar vulnerabilities from happening again.

Changes are made carefully, and the project is also constantly being fuzzed to discover new bugs.

If the intent is to run arbitrary, untrusted code outside a browser environment, `wasmtime` feels like the most secure option.

Without any optional features, a minimal executable linked against `libwasmtime` weights 22 MB, even if AOT is used and compilation functions are not called.

This is a bit more than `wasmer` (a basic standalone executable produced by `wasmer` weights 16 MB) but far less than `wasmedge`, that only ships a shared library that weights no less than 45 MB, which is as big as the `node` library (46 MB)!

To summarize, it's hard to strongly recommend one of these runtimes. You can't go wrong with any of them, unless your application needs one of their unique features.

JavaScriptCore

`JavaScriptCore` is to `Safari` what `V8` is to `Chrome`: a portable, full-featured JavaScript engine, backed by excellent, multi-level interpreters and compilers.

Both are used in the vast majority of web browsers we use today, but also, outside the browser in `node` (`V8`) and `bun` (`JavaScriptCore`), as well as in function-as-a-service services and as a way to extend applications.

And `JavaScriptCore` performance is excellent, often exceeding the one of `V8`.

Building on top of their existing compilers, they both eventually got support for WebAssembly.

Unfortunately, `JavaScriptCore` currently doesn't perform as well as `V8` for WebAssembly. As of today, if you're looking for a JavaScript+WebAssembly engine, I would thus recommend `V8`.

That being said, `JavaScriptCore` developers reached out to me, asking for a license file to be added to the benchmark files. So, it is very likely that they will fix the optimizations `JavaScriptCore` is currently missing, and the engine will eventually perform as well, or even better than other runtimes.

Wazero

To embed WebAssembly modules in a Go application, `wazero` should be your go to choice unless performance is absolutely critical.

`wazero` is entirely written in Go, with zero dependencies. That comes with many advantages: safety, portability, perfect integration with the language, and adding it to existing projects is trivial and adds negligible overhead compared to alternatives.

wasm2c

Now has come the time to introduce `wasm2c`, which is clearly in a different category.

Can a WebAssembly compiler include a WebAssembly runtime in order to run a WebAssembly version of itself to compile itself again? If that sounds confusing, go read how Zig got rid of the C++ compiler.

`wasm2c` is a WASM to C transpiler. The transpiler itself was written in a couple days, is super simple, fits in ~2000 lines of self-contained code and doesn't try to be smart nor optimize anything. It just literally translates individual WebAssembly opcodes into very dumb C code, and lets a C compiler take care of all the optimization work. It also includes minimal support for `WASI`.

The `wabt` package includes a similar tool, with more advanced features. But looking at what the simplest implementation could do seemed interesting.

Zero dependencies needed besides a C compiler (here, I used `zig cc` since it was already installed). The runtime overhead is… zero: there's no runtime! Memory overhead is also negligible. But how about performance?

Looking at this benchmark's results, performance of such a trivial approach turns out to be outstanding. In fact, it is neck-to-neck with the fastest WebAssembly runtime in every single test.

But there's a catch. The two main selling points of WebAssembly are portability (this is bytecode after all), and the guarantee that an application cannot access arbitrary memory locations.

Code compiled from `wasm2c`'s output provides the former: a single copy of WebAssembly code can run on multiple platforms.

However it doesn't currently do anything to enforce the latter. Doing so would require adding guard pages before the stack and after linear memory segments, or extra operations such as applying a mask to pointers used in load/store operations to keep them within a safe range.

But when the WebAssembly code to run is trusted, this very compact and simple approach works amazingly well, and was an excellent decision to bootstrap a compiler.

Leveraging this approach, w2c2 looks like a very promising project to keep an eye on (claiming to be only 7% slower than native code!)

Kudos, everyone!

A lot happened in the world of WebAssembly runtimes in the past two years. Projects were abandoned, but new projects saw the light, exciting new features have been added to existing runtimes and impressive performance improvements have been made (`cranelift` now being as fast as `LLVM`). There's no clear winner any more, only good options.

We also saw new applications of WebAssembly outside browser environments, and consistent ways to use WebAssembly across programming languages (Extism). WebAssembly also became an essential component for confidential computing (Vera Cruz, Enarx, Inclavare).

WebAssembly support in languages and tools also improved. Swift can now emit WebAssembly code. C/C++ code can be compiled to `WASI` using `zig cc` instead of manually patching `clang` or having to install yet another entire copy of `LLVM`. `tinygo` became more mature. And new programming languages targeting WebAssembly were born.

As to using WebAssembly in web browsers, and among other things, Cowasm was opensourced: an alternative to Emscripten developed by SageMath, with real-time collaboration features. This alone unlocks a lot of possibilities.

The future for the WebAssembly ecosystem definitely looks bright!

Jump back to the home page